Science of Computer Programming 75 (2010) 672–688

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Experience of building an architecture-based generator using GenVoca for distributed systems

Chung-Horng Lung *, Pragash Rajeswaran, Sathyanarayanan Sivadas, Theleepan Sivabalasingam

Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada

ARTICLE INFO

Article history:
Received 19 October 2007
Received in revised form 27 April 2009
Accepted 21 May 2009
Available online 31 May 2009

Keywords:
Generative programming
GenVoca
Software architecture
Prototyping
Distributed systems
Software patterns

ABSTRACT

Selecting the architecture that meets the requirements, both functional and non-functional, is a challenging task, especially at the early stage when more uncertainties exist. Architectural prototyping is a useful approach in supporting the evaluation of alternative architectures and balancing different architectural qualities. Generative programming has gained increasing attention, but it mostly deals with lower-level artifacts; hence, it usually supports lower degrees of software automation. This paper proposes an architecture-centric generative approach in facilitating architectural prototyping and evaluation. We also present our empirical experience in raising the level of abstraction to the architecture layer for distributed and concurrent systems using GenVoca. GenVoca is a generative programming approach that is used here to support the generation or instantiation of a particular architectural pattern in distributed computing based on user's selection. As a result, it can support rapid architectural prototyping and evaluation of both functional and non-functional requirements and encourage greater degrees of software automation and reuse. Lessons learned from the empirical study are also reported and could be applied to other areas.

1. Introduction

Software complexity that we face today becomes a great challenge in practice and the complexity still keeps increasing. To mitigate the complexity issue, we can increase software automation and raise the level of abstraction in programming [7]. Software automation and level of abstraction are not totally independent. Raising the level of abstraction in fact can facilitate software automation [6]. Software automation can be supported by various methods, such as generative approaches [15], program transformations using model-driven development (MDD) [10,11], metaprogramming [39], and program synthesis [5,25]. This paper proposes an architecture-centric generative approach with an aim to deal with the complexity issue.

The concept of generative programming has been postulated in the area of domain analysis and engineering since the 80's. Generative approach deals with modeling and implementing system families so that a particular system can be automatically generated from a specification written in domain-specific languages (DSLs). The main idea is that new system variants can be rapidly created based on a set of reusable assets or components. Generative programming, generally, is based on a common software architecture and the assets or components are at lower levels of abstraction than software architecture.

Software architecture captures the high-level structure of a system, among other things, that has tremendous impact on other software artifacts and non-functional requirements. Software architecture evaluation [22,26] is critical but practically challenging at early life cycles of software development primarily due to uncertainties of requirements and technologies. Architecture evaluation may involve re-engineering or restructuring of existing systems, which mostly is a time-consuming

* Corresponding author. Tel.: +1 613 520 2600; fax: +1 613 520 5727.
E-mail address: chlung@sce.carleton.ca (C.-H. Lung).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.05.003

manual process. Another realistic issue that often exists in enterprises is the tight schedule for software development. It is very difficult to effectively conduct thorough software architecture evaluation in practice under various constraints, especially with multiple software architecture alternatives. When multiple architecture alternatives exist, the complexity of evaluating both functional and non-functional requirements among them will increase significantly.

This paper advocates a generative approach to software architecture. The approach can help the architect rapidly or incrementally develop, and subsequently evaluate the software architecture effectively at early stages. Using the approach, evaluation effort can be dramatically reduced due to system generation rather than development from scratch. The architect can focus on the actual application, requirements, and/or spend more time on the quality aspects. The approach adopted in the research is to build a generator or a generative framework that consists of multiple architecture alternatives, so that the architect can choose and experiment with those alternatives for rapid prototyping, incremental development, or effective evaluation by collecting realistic data with executable systems. In addition to the benefit of rapid architecture prototyping, the complexity issue is mostly hidden from the user. The focus of this paper is to demonstrate the concept and the development of the framework instead of actual evaluation of final generated application systems.

Another objective of this research is to conduct an empirical study with existing generative techniques; namely, GenVoca [3,4,15]. We have applied the GenVoca generative programming approach directly to an industrial software system developed by Nortel. Specifically, the aim is to raise the level to software architecture generation by applying GenVoca to a real system. Raising the level of abstraction increases the degree that software development can be automated [6]. The process that we have adopted and the empirical lessons that we have learned could be useful in advancing software technologies in dealing with software complexity.

The rest of the paper is organized as follows: Section 2 briefly illustrates the background with emphasis on related experience, patterns and GenVoca. Section 3 demonstrates the modeling and development process, as well as the artifacts of the generator. Section 4 discusses related work. Section 5 depicts some critical lessons learned and future work. Finally, Section 6 is the conclusions.

2. Background

In this section, we briefly describe some challenges that we encountered in software re-architecting and architecture evaluation in distributed computing, which is followed by an overview of the related architectural patterns and the GenVoca generative programming technique [3,4,15].

2.1. Re-architecting experience of distributed and concurrent software systems

Distributed applications exist everywhere these days. Software patterns have become popular and well recognized in software engineering community [13,17]. Patterns in distributed and concurrent systems have been identified and documented [34]. Patterns in this area are classified into several categories. From the overall structure aspect, three patterns are common: Single Thread (ST), Half-Sync/Half-Async (HS/HA), and Leader/Followers (LFs); each one has advantages and disadvantages. In addition, those patterns are specialized and may not be easy to understand or implement for those who are not familiar with the problem area. This becomes a challenge to software architects in software architecture evaluation, especially for performance at the early stage.

Many software products have performance problems. Software Performance Engineering (SPE) [35,41] is vital for this issue. It is well accepted that SPE should be conducted early in the life cycle. Unfortunately, conducting SPE, especially at the early stages of the development, is difficult and requires a high skill level. To perform SPE effectively, the architect has to have intensive knowledge and experience in the application domain, software design, and performance engineering, which is uncommon in reality.

As an example, consider a real industrial case we examined. The project was a study of an advanced network traffic engineering technique to support load balancing, resource utilization, and path protection and restoration. When the project started, no suitable simulation tools were available and the network carriers requested an executable prototype for the proof of concept. A lot of effort was spent on software prototype solutions involving three and a half designers to facilitate the traffic engineering applications which had three engineers.

Further, because of the timing constraints, possible architecture alternatives were not carefully evaluated, which affected the outcome of the traffic engineering application comparisons. The results were mainly evaluated based on some network criteria such as packet loss, system utilization, and path restoration time for failures. An initial comparison of three network protocols showed that a newly proposed protocol at that time had more packet loss than an existing protocol, which was inconsistent with the original prediction. The software was later re-architected to support additional requirements without changing the high-level traffic engineering application, and the results for three different protocols improved up to about 20%, 3%, and 10%, respectively, for some scenarios. If the alternatives had been evaluated more carefully in the original design, there might not have been such variations in terms of performance gain, which was difficult to interpret consistently and would affect the comparison results of three network protocols.

We also learned from our re-architecting experience that it can be a time-consuming and error-prone task. In another case study, the system needed to be restructured to support additional QoS requirements. The existing system was limited due to its initial architecture: messages received from the network were processed non-preemptively. The Half-Sync/Half-Async (HS/HA) architectural pattern [34] was a natural fit for the restructuring. However, due to complicated concurrency

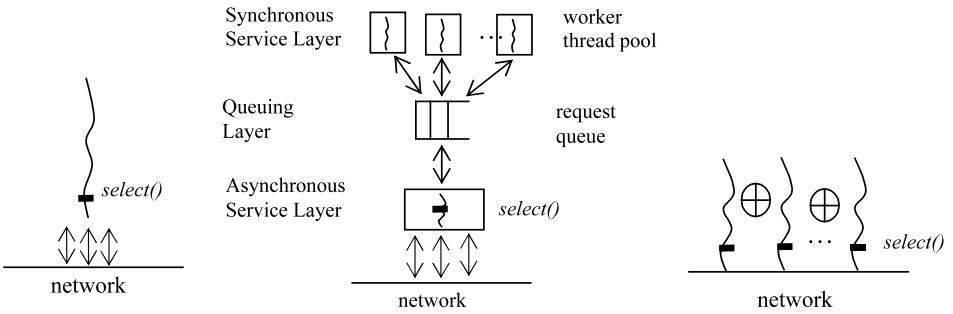

Fig. 1. Basic networked and concurrent architectural patterns [27].

controls and interactions with the application level, the restructuring itself took much longer than expected, even though the new design was well understood and documented in the pattern literature.

Our experience has motivated us to find a way to increase software automation and the degree of abstraction in order to cope with the complexity issue. The idea is to develop an approach that can be used to quickly generate working systems through the selection from a set of architecture alternatives. The designer can hence focus on the specific application and the architect can effectively evaluate architecture alternatives by analyzing concrete data obtained from those working systems.

2.2. Architectural patterns in distributed and concurrent systems

Patterns at different levels of abstraction have been adopted in our generator or generative framework. All those patterns are documented in [34]. This section emphasizes those related architectural patterns. Some lower-level design patterns used in the framework are not described in this section. Those patterns include Monitor Object, Scoped Locking idiom, Reactor, Connector, and Acceptor. Interested readers may refer to [34] for detailed explanation of any patterns of interest.

Fig. 1 demonstrates these three alternatives adopted in the research at the abstract level. As stated in Section 2.1, the original software system was developed using the ST approach, as shown in Fig. 1(a). Enormous efforts were spent to re-engineer ST to either HS/HA, Fig. 1(b), or LFs, Fig. 1(c), for additional QoS requirements and performance improvement [1, 28,29,42]. The only thread in the ST approach will both handle events via the `select()` function and process the incoming messages arrived from the network. However, this approach could have a performance concern. Specifically, when a message is being processed by the thread, the arriving messages have to wait until the thread finishes executing the message.

The situation may be improved using either the HS/HA or LFs pattern as the overall architecture. HS/HA divides the system into three layers, as shown in Fig. 1(b). The asynchronous layer reads messages and stores them in the queuing layer. Multiple worker threads will read messages from the queue in a synchronous fashion and handle those messages subsequently.

In LFs, multiple threads are running concurrently, each thread functions similarly to that in ST and synchronization of those threads is provided. However, only one thread at a time – the leader – waits for a network event to occur. Other threads – the followers – can queue up, waiting for their turn to become the leader. Once the leader detects an event, it promotes one of the followers to be the leader. It then becomes a service-processing thread.

2.3. Generative programming using GenVoca

Generative techniques have been intensively discussed in the area of domain analysis and domain engineering. More applications have adopted the generative approach, such as simulation [26], automotive industry [8] and GUI [14]. We also have conducted domain engineering and developed a generative framework in Java for distributed and concurrent systems [30]. This paper follows the same idea as our previous work: building an architecture-centric generative framework to support rapid architectural prototyping and/or architecture evaluation. The key difference is that we have directly applied the GenVoca technique [3,4,15] to a larger scale of (∼30 K) C++ LOCs, not including GUI and third party software, and complex network emulator software developed by Nortel.

The purpose of generating programming is to replace manual adaptation and composition of components with automatic configuration and generation of components on demand. Generative programming consists of three elements: problem space, solution space, and the configuration knowledge that maps from the problem space to the solution space [15]. The problem space deals with modeling of the problem area, including domain-specific concepts and features. The solution space provides elementary implementation components and a common system architecture. The configuration knowledge specifies construction rules, dependencies, and feature compositions. Program generators can support the configuration knowledge construction. Generators may be implemented using different technologies, such as preprocessors, application generators, built-in capabilities of a language (e.g., template metaprogramming in C++), algebraic specifications [24], etc.

This work presents an experimental study of building a generator or generative framework using GenVoca. GenVoca was chosen primarily because the system and its variants under study were developed in object-oriented C++ and GenVoca is particularly useful for generating object-oriented models [15]. Next, based on a study [36], GenVoca model requires less

parameterized components and exhibits lower level of code duplication than that of library components. Further, GenVoca is easily accessible and is a proven technique that has been adopted in various applications, such as databases, avionics, and network protocols [4]. On the other hand, most reports using generative programming and GenVoca dealt with lower-level components and were based on a common architecture. This paper investigates the applicability of GenVoca to architecture generation, which has the potential to significantly decrease development time.

GenVoca is a compositional technique that is used to build generators by assembling layers of abstraction. GenVoca implements higher-level modules by assembling lower-level components. The main idea is to compose objects or a system out of layers. Each layer deals with a specific feature. Each layer contains a certain number of object classes and the layer at a higher level adds new classes or methods to the layer below it via parameterization to combine and/or customize components.

The following highlights the main steps involved in building a generator using GenVoca [15].

- Identify layers of abstraction. More specialized layers are at higher levels and more general ones are at lower levels.
- Treat each layer as a parameterized component and a given layer can access the classes contained in the parameter that reside in the layer below the current layer.
- Provide alternatives and parameterized layers. In addition, this step also specifies the rules for composing layers.

In the GenVoca model, each feature is represented as a separate layer. Features are identified through a modeling phase that factors out commonalities and variabilities. Concrete components are defined by type expressions illustrating layer compositions. Each layer contains classes, attributes, and/or methods implementing the corresponding feature(s). For instance, the following type expression [15]:

```
bag[concurrent[size_of[unbounded[managed[heap]]]]]
```

represents six layers. The expression specifies a `concurrent`, `unbounded`, `managed` `bag` which allocates the memory for its elements in the `bag` from the `heap` and counts the number of elements (using `size_of`) in it. The bottom layer, `heap`, implements a generic memory allocation scheme that can be used by various data structures, e.g., bag and queue. The `managed` layer deals with the allocated memory using a free list; `unbounded` provides a resizable data structure obtained from the `managed` layer; `size_of` adds a counter feature; `concurrent` handles code serialization; and, lastly, the top layer, `bag` implements specific bag operations for unordered collection of objects.

Section 3 illustrates GenVoca with a concrete case study, including feature modeling, on communication software systems. A thorough discussion of GenVoca can be found in [15].

3. Modeling and development process using GenVoca: A case study

The main objective of the generator is to instantiate a specific system using one of the architectural patterns in distributed computing, i.e., ST, HS/HA, and LFs, selected by the user. The generator is developed based on a study of existing Client/Server (C/S) and Peer-to-Peer (P2P) systems. The C/S system was originally developed for a course project. It is a simplified transaction-oriented system, but has been thoroughly tested by different groups [1,23]. The emphasis of this study, however, is on the communications and messaging aspects. The P2P system, CgNet, was a network emulator developed by Nortel [20]. Section 3.1 describes the system in more detail. Building the generator using existing robust systems rather than reinventing the wheel reduced development time.

The original CgNet was designed using the single thread approach. A lot of efforts were spent to re-architect the original system using HS/HA [29] and LFs [1]. The generator was then constructed via reverse engineering of existing systems and subsequent forward engineering of building reusable components, GenVoca layering structure, and the corresponding implementation. The following highlights the main tasks that were involved in re-engineering process.

- Reverse engineering of existing systems: This step is essential for understanding both the problem space (communication networks and application software) and solution space (networked and concurrent software) as the framework was not built from scratch. Both distributed computing models (C/S or P2P) were studied; each model was implemented using those three patterns (ST, HS/HA, and LFs). All the six alternatives (2×3) were studied and compared. Main features in each alternative were identified.
- Forward engineering: A feature model that comprises those six alternatives was developed. Commonalities were factored out and variabilities were identified for the construction of a GenVoca layering structure which was followed by the actual implementation, including components and the domain-specific language.

The following sub-sections describe each phase in more detail with concrete examples.

3.1. Reverse engineering

Reverse engineering has been widely applied in practice. What distinguishes this work from others is the emphasis on variability analysis and modeling from the architecture level to lower levels. The main tasks of this phase are:

(1) Identify variation points, starting from the architecture level down to lower levels, and potential variations under each variation point.

(2) Identify components/features and their relationships for each architectural variation.

Table 1
Abstracted features from the Client/Server systems

ST	HS/HA	LFs
Socket setup	Socket setup	Socket setup
Initial buffer setup	Initial buffer setup	Initial buffer setup
Thread creation • Main thread creation	Thread creation • Main thread • Worker threads	Thread creation • Leader/Followers threads
No queue	Queuing layer (one queue)	No queue
No thread management	Synchronize worker threads and the main thread, but no Join/Promote thread management	Need Join/Promote thread management, no main thread synchronization
Message processing: Exchange messages and process data	Message processing: Exchange messages and process data (tied to synchronization of threads)	Message processing: Exchange messages and process data (tied to thread management)
Main functionalities: Insert/ Retrieve/Remove	Main functionalities: Insert/Retrieve/ Remove	Main functionalities: Insert/Retrieve/ Remove
Application initialization	Application initialization	Application initialization

Table 2
Main functions of CgNet's components

Component	Function
Node	A network device that is used to construct a route for each destination node and forward packets to the next hop based on the route. A router node consists of the following processes.
Generator	Randomly generates data packets that will be sent to other routers.
Sink	Consumes data packets received from other routers or its own generator.
Statistics sink	Consumes statistic reports generated periodically by the node process.
ONC	Automatic network traffic controller for load balancing, protection and restoration, and specific QoS requirements.
Manual controller	Network operator interface that is used to send commands to node processes.

To meet our main objective to support rapid architecture generation, the variation point starts at the architecture level. As stated earlier, two models (C/S, P2P) and three architectural patterns or variations (ST, HS/HA, LFs) for each model were studied in distributed computing. These three well-known alternatives, ST, HS/HA, and LFs have been selected mainly due to their popularity. More variations can be added later if new patterns are identified or other solid design is developed. The basic architecture of these three variations is shown in Fig. 1.

In addition to the alternatives at the architecture level, possible variation points at lower levels are identified. Lung [27] conducted a preliminary variability analysis for various architectural alternatives in communications software for the server process based on those patterns. A selected architectural alternative may consist of more variation points and each variation point in turn can be realized with different variations. The queuing layer, for example, could consist of a single queue or multiple queues, each for one priority group or quality-of-service (QoS) requirement. Another simple example of variations at the low level is the number of threads in HS/HA or LFs.

Although these three alternatives are architecturally different, they share common lower-level features and components. This phase also involves identification of components and features, and their commonalities and differences and relationships or interactions. A feature in our context is similar to a software functionality. The selection of one variation or feature may enable or disable the selection of other features or variations. For instance, selecting HS/HA means that a queuing layer, as shown in Fig. 1(b), is required. Feature modeling is a crucial step to generative programming or GenVoca. Table 1 shows the main features derived from the C/S systems. This is a basic step in feature modeling.

For CgNet, it is much more complicated. CgNet is a P2P system that was developed to emulate real network traffic and to study a new protocol for the traffic engineering service. Fig. 2 demonstrates the overall structure of CgNet. The network consists of a central controller that is used for network management and a set of peers or `Routers` (see Fig. 2); each `Router` contains a group of software components. Table 2 lists the software components and summarizes their main functions. Each component can be further decomposed and examined. The main component that was re-engineered was `Node`, as depicted in Fig. 2. A `Node` process has around 10K C++ SLOCs. Other components are related to the traffic engineering applications. `Node`'s main functionalities include setting up communications with peers, creating a routing table based on shortest distance, receiving and processing messages from other destinations.

Even though the original CgNet was not developed directly based on patterns, it was not difficult to identify similar concepts. Specifically, `Reactor`, `Connector`, and `Acceptor` design patterns were used in the original CgNet. Modeling of those patterns was straightforward in our case, since they have been well documented in the patterns community and it was not difficult to identify similar functional behaviors in CgNet as CgNet is in the same problem space and it was developed

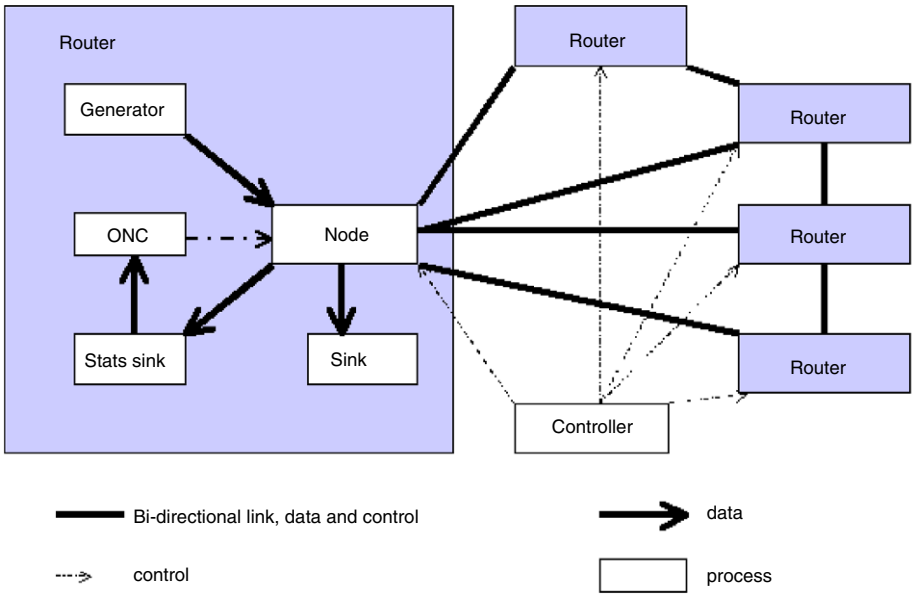

Fig. 2. System structure of CgNet.

Table 3
Abstracted features from CgNet (P2P) systems

ST	HS/HA	LFs
Socket setup	Socket setup	Socket setup
Thread creation • Destination • Statistics	Thread creation • Destination • Statistics • Worker threads	Thread creation • Destination • Statistics thread • LFs threads
No queue	Multiple queues for QoS	No queue
No thread management	Synchronize worker threads and the main thread, but no thread management for Join/Promote	Need Join/Promote thread management, no main thread synchronization
Main functionalities: Perform network emulation tasks	Main functionalities: Perform network emulation tasks (tied to synchronization of threads)	Main functionalities: Perform network emulation tasks (tied to thread management)
Other functionalities: Supporting functionalities	Other functionalities: Supporting functionalities	Other functionalities: Supporting functionalities
Application initialization	Application initialization	Application initialization

by experienced designers. Additionally, two more lower-level design patterns, `Monitor Object` and `Scoped Locking idiom` [28], were identified suitable for concurrency control and were added to the systems. However, this step extends the traditional software modeling by incorporating multiple architectural variations and variations at lower layers. The step shares concepts with the product line architecture modeling, which will be elaborated more in Section 4.

Similar to the analysis conducted on the C/S model, the features of CgNet and its representative variants were identified as shown in Table 3.

3.2. Forward engineering

This phase mostly follows the traditional feature modeling process presented in [15,21]. Here we illustrate it with concrete and more complicated examples. One key objective is to raise the level of abstraction to the architecture layer with an aim to increase the degree of automation and support front-end software architecture tasks, such as evaluation and concept demonstration.

A critical task in this phase is to construct a feature model based on the similarities, variabilities and their dependencies in a family of systems. The feature model serves as a blueprint that captures identified variation points and variations. Based on the feature model, we then need to identify components that will be implemented. Another crucial task in this phase is to design a common architecture for those implementation components. The feature modeling step also identifies feature/component composition rules. In other words, selecting one feature may enable or disable the inclusion of another feature. The subsequent construction of a framework using GenVoca is closely tied to parameterization techniques.

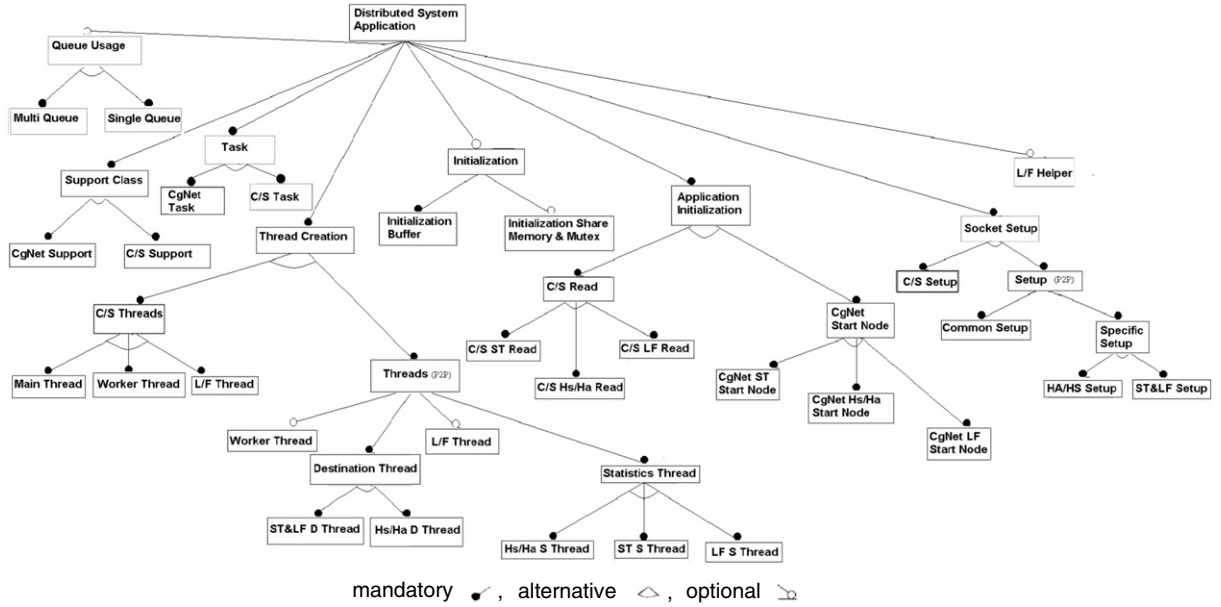

Fig. 3. Feature diagram for Client/Server and CgNet.

The following points describe the phase in more detail, step by step.

(1) Conduct feature modeling. This step extends the features shown in Tables 1 and 3. A feature model captures similarities and variabilities of systems in a particular domain or for a concept. Each feature can be mandatory, alternative, optional, or or-feature [5,16,21].

As shown in Fig. 3, the distributed computing is the target problem, and it has `Thread Creation`, `Application Initialization`, `Socket Setup`, `Task`, and `Support Class`, as mandatory features. This implies that any distributed application should have these variation points based on previously identified features. Furthermore, the `Socket Setup` variation point demonstrates alternative features for C/S and P2P socket setups. It also shows that P2P has a common socket setup feature (`Common Setup`) independent of the patterns and some specific socket setup alternatives (`HA/HS Setup` or `ST & LF Setup`) depending on the particular pattern selected. In other words, socket setup for P2P requires a common feature (`Common Setup`) and a specific feature, either `HS/HA Setup` or `ST & LF Setup`.

Likewise, other features listed earlier are visible on the diagram. As an example, the `Thread Creation` variation point is featured as a mandatory component in the diagram, but it also has subsequent alternative variation points, which emphasizes that the characteristic of C/S or P2P system varies. Moreover, the diagram's `C/S Threads` variation point is decomposed into alternative variations, which indicates that only one component at the bottom layer can be selected at a time depending on the design pattern. On the other hand, the `P2P Threads` variation point for CgNet combines optional and mandatory variations, which describes those components and corresponding compositions that are needed to support the required features. (ST is the default choice which is not shown in the diagram.)

(2) Identify Implementation Components from the Feature Model. Implementation components are those placed at the bottom of the tree or leaves in the feature model. Fig. 4 illustrates the implementation components identified through the feature model developed based on the existing systems we studied. Each component can be further modeled in detail.

(3) Design Common Architecture for the Implementation Components. The proposed framework consists of multiple architectural alternatives for the target domain; however, the architecture of the framework itself is a layered structure. The idea basically follows GenVoca which makes use of the layering technique. This architecture is composed of the parameterized components and standardized interfaces to compose objects from layers. Furthermore, each layer handles a specific task. Before putting the components into layers, the dependency of the components needs to be analyzed. Similar components are put into the same layer and components dependent on other components are placed at higher layers.

This step is crucial and the technique was adapted to the non-trivial distributed applications. Fig. 5 presents the layer architecture for the distributed systems analyzed. In our case study, six layers are required for CgNet and P2P. The top layer is not needed for the C/S application, since there is only one server needed in the system.

(4) Implement Parameterized Components. This step makes use of C++ templates intensively. Detailed discussion can be found in [15]. Appendix A illustrates the structures defined for CgNet using three different patterns. Note that the structure corresponds to the layer architecture depicted in Fig. 5. For instance, the following structure shows that in order to start the server of the C/S model using ST, four components are used, which are also highlighted in Fig. 5.

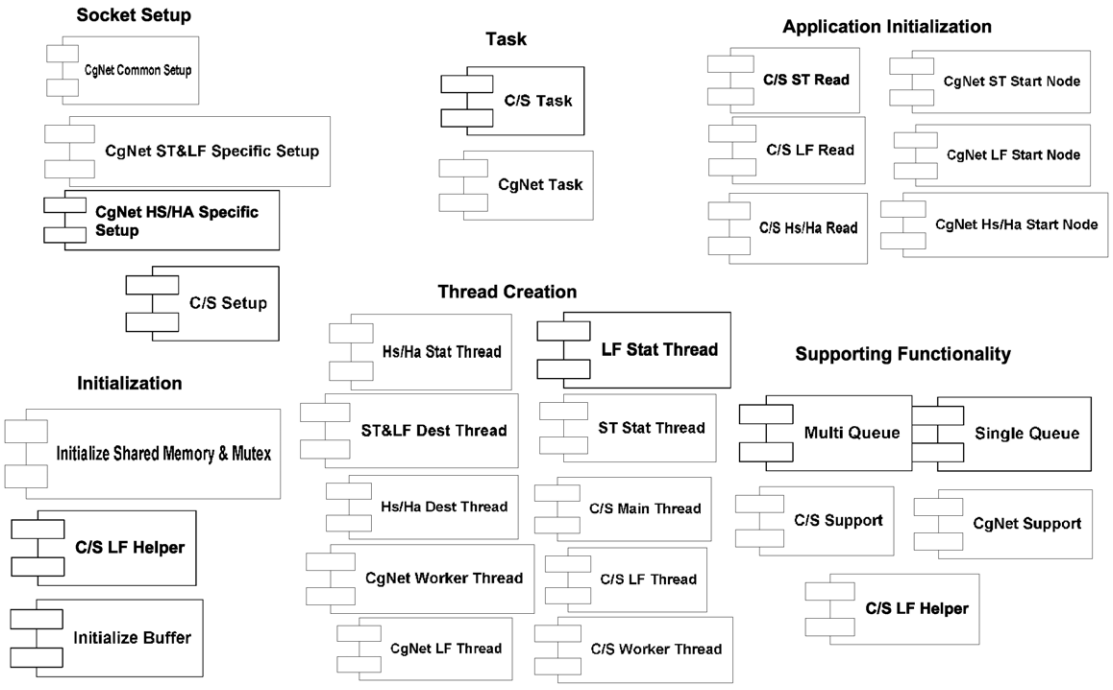

Fig. 4. Components for distributed and concurrent systems.

Layer 6: Start Node ST | Start Node LF | Start Node HS/HA
Layer 5: Setup P2P Common Setup | Create Worker Thread LF
Layer 4: Main Thread HS/HA Read | **Main Thread ST Read** | Create Worker Thread HS/HA | Do Read LF | P2P HS/HA Specific Setup | P2P ST&LF Specific Setup
Layer 3: **Shared Memory and Mutex Setup** | **ST Initialize buffer**| ST&LF Destination Thread Creation | HS/HA Destination Thread
Layer 2: Initialization Buffer Others | ST Other Thread Creation | LF Other Thread Creation | HS/HA Other Thread Creation
Layer 1: CgNet Task (P2P) | **Client Server Config**

Fig. 5. GenVoca layer architecture for distributed systems.

```
struct CS
{
typedef
MainThreadSTRead<SharedMemoryAndMutexSetup<STInitializeBuffer<ClientServerConfig>
> > createThreadST;
};
```

(5) Organize Configuration Knowledge for the Implemented Parameterized Components. In order to come up with a particular system there should be a mechanism to assemble the required components and produce the particular application required. Configuration knowledge completes the specification and assembles the components to build the specific application.

The component assembly can be done in two ways, manual assembly and dynamic assembly [15]. This paper shows the dynamic assembly. A domain-specific language was defined as presented in Appendix B. The language defines the distributed models and patterns used in the domain. A generator using templates is then defined and shown in Appendix B. In addition, how to instantiate an instance dynamically is also specified. The syntax and semantics of the domain-specific language and

the generator are clearly presented in [15]. The paper does not elaborate them here. Interested readers may refer to [15] for detailed descriptions.

3.3. Evaluation of the proposed framework

Evaluation of a framework is an important task, as it is desirable to generate high-quality executable systems. One possible concern of the proposed approach is the qualities of the framework itself. The framework and all its enclosed specific architecture alternatives should be carefully evaluated both functionally and non-functionally before it is used. The concept is a concern in reuse in general, i.e., before a component or a system is reused, it should be actually used and/or thoroughly tested. From the functional perspective, the framework should have the same behaviors as those of individual standalone systems built using various architectural patterns. From the non-functional perspectives, the main concerns include performance, expandability, and maintainability.

One strength of our framework is that it was built using existing operational systems and well-documented software patterns. CgNet has been built and tested for some years and several research projects or variants [1,23,28,29] have been built upon it. Those variants have been carefully tested with respect to functional and performance requirements and have been reported previously.

We have also compared the generated systems using the framework with the original ones based on functional testing and performance testing. We have tested the functionalities of the applications generated by the framework against the original baseline systems [33]. In addition, the DSL for the generator (see Appendix B) has been carefully tested by generating and testing each of the six architectural alternatives.

For the performance aspect, because the operations in GenVoca are usually defined inline, calling a method using an inline definition does not incur any extra overhead. With GenVoca models, we achieve clear separation of layers without additional performance penalty. The following simplified code segment presents a demonstration. More detailed description related to the DSL and the generator is illustrated in Appendix B.

```
1.  If (systemSelection == cgNetSingleThread) {
2.      typedef FRAME_WORK_CGNET<CgnetEmulator<ST> >::RET CG_ST;
3.      CG_ST cgST;
4.      cgST.startNode ();
5.      return 0;
6.  } else if (systemSelection == cgNetHalfSyncHalfAsync) {
7.     typedef FRAME_WORK_CGNET<CgnetEmulator<HSHA> >::RET CG_HSHA;
8.     CG_HSHA cgHSHA;
9.     cgHSHA.startNode ();
10.    return 0;
11. }
```

Line 1, as depicted in the sample code, checks if the user selects the CgNet single thread alternative. Line 2 is a type expression that defines a type `CG_ST` (CgNet single thread). `FRAME_WORK_CGNET`, as illustrated in Appendix B.4, is a template structure that consists of GenVoca layers. Line 3 instantiates a cgST object. The first three lines are extra overhead compared to the original independent baseline system using the single thread. But these extra lines are only used in the initialization. Once an architectural alternative is selected through the user interface, the rest of the implementation becomes very similar to the original single thread system and the execution does not introduce extra performance hit. The method `startNode()` on line 4 is defined in the `startNodeST` class; `startNodeST` is a layer 6 feature shown in Fig. 5. Similarly, lines 6 through 10 illustrate the case for the selection of the HS/HA pattern, definition of class (CG_HSHA) via parameterization (line 7) and instantiation of an HS/HA object (cgNSHA on line 8). `cgHSHA.startnode()` on line 9 invokes a method that is defined in `startNodeHsHa` which is a layer 6 feature depicted in Fig. 5.

As a quantitative illustration for the performance evaluation using the generator, Table 4 presents a performance comparison for the LFs architecture with and without the generator. As depicted in Table 4, the performance differences of the original LFs system without using GenVoca and the system generated using the GenVoca generator are negligible. For this particular case study, 9 threads were instantiated for the LFs pattern and the execution time was 15 min for each data point. The traffic rate at 100 is the base packet generation rate (the highest rate where no packet lost is experienced) specifically configured for CgNet. The performance evaluations for ST and HS/HA with and without the generator reveal very similar results; hence, they are not presented here. The evaluation results demonstrate that the performance of the systems generated with the framework is almost identical to that of the original systems.

The current framework only has three commonly used architecture alternatives based on known solutions. Ideally, a tool for architectural prototyping should provide a complete set of alternatives. In this case study, this is a concept demonstration. In addition, these three alternatives are representative solutions in this domain.

The framework can be further expanded by adding more proven architecture alternatives, if available, or other features. For instance, numerous QoS scheduling algorithms have been reported in communication networks community. The queuing layer is another potential area that can be expanded by adding QoS algorithms to it, so that users can simply select a certain algorithm and monitor the behavior or compare different algorithms.

Table 4
Performance comparison between LFs Systems with and without the generator

Traffic rate	CgNet with leader/followers architecture without generator		CgNet with leader/followers architecture with generator	
	Packet processed	Packet discarded	Packet processed	Packet discarded
100	2352980	0	2352988	0
150	3210032	105676	3210028	105668
200	4079610	290166	4079596	290162
250	4848298	525296	4848298	525292
300	5531919	837522	5531912	837512

The application layer can be replaced with another specific application. For instance, the traffic engineering layer in CgNet can be substituted with another Web application. Generally, the framework is limited to a specific problem domain. Nevertheless, effective domain engineering is advocated for specific areas. For different areas, a different framework may have to be built, and this is usually time consuming. Therefore, the proposed approach is better suited for situations where there is a need to repeatedly instantiate a system or for third party evaluators who provide services to other companies or organizations.

As described earlier, the framework can facilitate rapid architectural prototyping and/or architecture evaluation by quickly generating working systems based on different architectural alternatives. Another option to support prototyping and evaluation is to use three independent baseline systems, ST, HS/HA, and LFs separately. However, these three separate systems share commonalities as demonstrated in the feature modeling step. Clearly, maintaining three independent systems becomes inefficient in terms of evolution cost and scaling; that is, maintaining common components or adding new features may significantly increase the amount of work. By conducting feature modeling and applying the generative technique, we can avoid or decrease code duplication by means of inheritance and templates, which reduce the maintenance cost as well.

4. Related work

Software architecture has been recognized as an essential factor in successful software development. Practicing architects, unfortunately, do not have many tools to support front-end analysis. Our main point is to build a framework that can be used to instantiate different systems with different software architectures. The generated systems can facilitate the evaluation of architectural alternatives against a set of criteria, including both functional and non-functional requirements.

The concept of our approach is not totally new. In addition to the generative programming technique and GenVoca, similar concepts have been discussed in various areas. The following discusses some closely related work.

Component-based software engineering (CBSE) [18] is related to our approach. In fact, our framework also consists of identification and construction of reusable components. One main difference between CBSE or even a catalog of well-documented patterns and our proposed approach is that we look at the problem from the architecture perspective. CBSE does not address software architecture explicitly. There may be many ways to glue the components. Without an architecture, composition of components may not be trivial. Some CBSE approaches also address composition of components. The main difference between those approaches and our method is that we also explicitly identify components that can be used across some or all architectural variations and subsequently we construct reusable components based on the analysis.

Model Integrated Computing (MIC) [37] has produced fruitful results in system synthesis in embedded applications. In this area, domain-specific modeling environments are provided and the software and hardware system building blocks and their composition methods are well defined. Application developers can simply use this approach to facilitate analysis of the models and to automatically synthesize applications from the models. However, MIC has less success in areas of classical software applications [5].

AHEAD [4] extends GenVoca to express diverse representations, such as code, rule, and makefile. AHEAD offers an infrastructure to modularize problem domains by features and a mechanism to generate applications by composing features. AHEAD can be used to support the modeling step for components identification and composition. Their objective is different from ours. AHEAD does not explicitly discuss multiple architectural alternatives and the application to software architecture evaluation and prototyping. The AHEAD paradigm is primarily developed from the programming perspective, though it may be used to support our objective like GenVoca.

The concept of product line architectures [6,12,32] is another area that shares similarities with our approach. Typically, a product line approach starts with some kind of base product architecture which is used to establish the base for the commonality and variability analysis. The base product is then expanded by incorporating domain information to produce product line architecture and the variant is called product architecture. However, the variabilities for product line architecture are often captured at a lower level than architecture, e.g., component, class, method, and even variables. On the contrary, the framework developed using our approach could consist of multiple architecture alternatives; some could be very different and each architectural variation is equivalent to the base product architecture in the traditional product line approach.

Model-driven development (MDD) is an emerging approach that also focuses on high-level specifications of programs and automation in software development. MDD uses models to represent a program. A model is specified in a DSL that is

particularly developed to capture details of a target problem area [6]. MDD uses model transformations to convert platform independent models (PIM) to platform-specific models (PSM) [38]. Model transformation is a specific generative technology. In addition, most current MDD efforts focus on platform independence. Generative programming addresses systematic domain variability management and development of DSL which is closely related to MDD.

Metaprogramming is proposed to raise the level of abstraction in software development with large pieces or components. In metaprogramming, program development and synthesis is a computation [38]. Generative techniques deal with metaprograms that synthesize other programs. Trujillo, et al [39] postulates the concept of generative metaprogramming which is an approach to metaprogram generation that will synthesize a target program of a product line. The main idea is to accelerate the development of metaprograms by generating them rather than implementing them from abstract specifications.

Overall, the approach proposed in this paper is distinguished from most other generative approaches in that their aim is primarily to generate a system or develop a program—in some cases, the "final" system. Our framework is primarily used to support rapid or incremental architecture development by providing executable systems (infrastructure rather than the final system), using different architecture alternatives from which the architect can build specific applications. The approach can support the comparison of architecture alternatives and evaluation against quality attributes by instantiating working systems using different architectures. Architecture tradeoff or sensitivity analysis [22,26] has been discussed extensively. One challenge in this area is that more specific information is often needed to provide more precise evidence for various quality attributes. In practice, however, the information may be difficult to obtain without a similar working or executable system. In other words, our approach can have a complementary role by providing quantitative or more concrete information to support architecture tradeoff or sensitivity analysis, especially in performance, scalability, and availability.

For new systems, stakeholders may not have a clear or consistent understanding of requirements, the operational approach may also facilitate discovery of unanticipated quality attributes. In our case study of the traffic engineering application, CgNet, the quality attributes initially identified were performance, scalability, and availability. The system was originally developed on Linux and was slightly modified without much difficulty for the Sun machines for another group. However, many problems were uncovered when the system was experimented on ChorusOS. As a result, an adaptation layer was added to the original design to support portability. With an executable system, it is also easier to evaluate the openness or constructability of a system by actually adding or modifying some components. These factors may also be considered as important quality attributes for some situations or stakeholders.

Based on our experience, it is often necessary to actually build a low-cost executable system that reflects the critical architectural elements and qualities of the target system [30], which is also the main theme of architectural prototyping advocated by Bardram et al. [2] and Martensson et al. [31]. Architectural prototyping allows the architect to explore different alternatives and receive concrete feedback, which could provide valuable information for balancing qualities and evaluating architecture. The difference between our approach and that of [2] or [31] is that our method takes one step further by providing a framework that consists of commonly used and well-accepted built-in architectural alternatives. The architect can select an option and instantiate an executable system quickly for comparison, exploration, and learning.

Although ArchE (Architecture Expert Design Assistant) [9] and our approach are not identical, they do share a main goal; namely, both are architecture-centric tools to assist the architect. Currently, ArchE is not used to generate a working system. Instead, ArchE is a general-purpose tool as it can provide guidelines and techniques to support quality assessment of systems in various areas. General guidelines can be applied to a wide range of applications, but there may be gaps between guidelines and actual realization. Our approach, on the other hand, provides specific and executable solutions by adopting an existing proven design in a specific domain. The applications are limited to the target areas that the framework is built for. However, the working solutions can be used to collect quantitative or realistic data, as discussed in Section 1, which usually have more direct impact. The idea is to aid the software architect in the early stages to examine the solutions generated by the framework for requirements validation and architecture evaluation.

One frequently asked question is the effectiveness of such a framework compared with modeling techniques. The proposed framework is not meant to replace modeling or discount the benefits of modeling. Modeling includes software modeling, e.g., UML, and performance modeling. With respect to software modeling, we actually used UML for each variation separately, i.e., traditional software modeling is part of the process. On the other hand, there is still a need to explicitly model the variability over multiple layers of abstraction. In our study, we also explicitly identified commonalities and differences for components that are used for different patterns. In other words, some patterns are common to all three architectural alternatives, even if the high-level descriptions are quite different.

Another point is that the proposed framework needs to be modeled only once and then it can be repeatedly used. In other words, more effort needs to be spent on the front-end analysis by incorporating multiple alternatives into a framework and identifying commonalities and variabilities of components. But the effort is mainly spent only once. Again, the concept is similar to domain analysis or development for reuse.

Performance modeling is a critical ingredient to SPE. From the performance modeling perspective, the framework can play a complementary role. Performance modeling, e.g., Layered Queuing Networks [40] or Stochastic Process Algebras [19], can be adapted more easily to a variety of applications and is useful for scalability analysis. On the other hand, each performance modeling technique has limitations, such as modeling of lost packets, failure scenarios, or state explosion. Secondly, performance modeling often depends on realistic estimations, such as execution time or probabilities for diverse decision points for different execution paths, and it may be difficult or time consuming to obtain these for complicated

system interactions within an application or between the application and the computing resources. By quickly generating a working system, the data from various working prototypes can be measured much more efficiently and precisely. The data could be stored in a performance knowledge base [41] and fed into performance models for further sensitivity or scalability analysis. Woodside et al. [41] also advocate describing the system with different values of factors, including variations in design.

5. Lessons learned

This paper reports our empirical experience of applying GenVoca to architecture-level program generation in distributed computing. The concept of building a generator using GenVoca is not new. Nevertheless, several lessons have been learned from this experiment.

One main purpose of this study was to investigate the applicability of GenVoca to support the concept of architecture-centric software generation to cope with increasing software complexity. Specifically, we conducted real experiments using GenVoca to build a generator for high-level software artifacts; in our case, different architectural alternatives. The result demonstrated that it was feasible to apply GenVoca to the modeling and generation of variations at the architecture level. As a result, the degree of automation was increased with variations at this level. Moreover, the approach enhances the leverage from both architectural prototyping and evaluation perspectives, as executable systems using distinct design alternatives could be generated to support subsequent analysis or comparison.

A key concept advocated in the paper is the modeling of variation points and variations at various levels. Identification of variations at different levels is not straightforward and is time consuming. For our case study, some key variations based on software patterns have been publicly documented and we have previously built systems using relevant patterns, but the modeling and synthesis task still consumed a lot of efforts. Variations (regardless of the abstraction level) may not be easily identified and additional testing endeavor was required to cover different variations, possible combinations of variations, and the domain-specific generators. Another issue of variability modeling is the granularity level. Or, to put in another way, commonalities or variations can occur at a very detailed level. For instance, the top layer of our case study has three variations to start a node. After a detailed analysis, we have found out that these variations, in fact, still have subtle similarities at a very fine-grained level which can be further factored out. To conduct variability analysis at that level can even increase reuse across architectural alternatives, but it requires more efforts.

Generative programming is not just programming as the term may indicate, it actually is `generative software development` that includes changes to the software development process and development of various software artifacts. The process involves systematic domain scoping, domain analysis, domain engineering, design, and implementation. Although the ultimate goal of generative programming is to produce code, generative programming requires producing various sorts of non-code artifacts, including models (e.g., feature model), domain-specific generators, documentations (design and tesing), and so on. The process is non-trivial. The analysis and modeling phase is essential to the end success, even if GenVoca deals with the code level.

The initial learning curve of GenVoca was high and C++ template programming and debugging required highly specialized skills. Building such a framework, in practice, also needs to consider the cost issue which is out of the scope of this research. The study was based on the concept of design-for-reuse to support further rapid architectural prototyping or architecture evaluation of alternatives in the problem area. Therefore, the proposed approach is more cost effective if the framework will be used multiple times.

GenVoca expresses software at the code level. Raising the level of abstraction for software representation is a trend in software development and can increase the degree of automation [6]. Various technologies are converging, particularly, MDD, metaprogramming, product lines, and software architecture, which raises several interesting questions, including:

- Integration of top-down generative approach and bottom-up component composition method: The framework was built upon existing working systems explicitly using different architectural alternatives. Consequently, variability analysis and modeling became easier than building from the beginning. Realistically, this step may require more effort either from the re-architecting or the forward engineering perspective. The framework, from the usage point of view, was a top-down generative approach. However, the development of the framework was strongly tied to bottom-up component-based software engineering. Overall, architectural variations served as blueprints for component compositions, while components facilitated system assembly, evaluation, and evolution. Software development process needs to be tailored to support the idea of building such an architecture-centric framework. In other words, the top-down generative approach and the bottom-up component-based software engineering also need to be integrated into the development process.
- Component qualification and adaptation: The framework was constructed through re-engineering with robust working systems, component qualification and adaptation tasks, hence, were relatively simpler. In practice, component qualification could be time consuming and the component adaptation aspect also could take time for quality or verification reason if components are developed from scratch.
- Research on model or program transformation: A program can be specified with one model and then be transformed to another model or a lower-level representation. Model transformations need specialized support in different perspectives, including system modeling and software transformation. Model transformations have become an important topic.

A model, in this context, could also mean a pattern. For instance, a software system developed using the ST approach may need to be transformed to HS/HA as requirements or technologies evolve. Manual re-architecting effort is time consuming and error prone as reported in our previous study [29]. Those three architectural variations have been used and well documented; similarities and differences are well understood. Therefore, domain-specific software transformation tools could be built to facilitate software evolution from one architectural variation to another.

Detailed performance evaluation of various architectural alternatives is not the focus of this paper, as it depends heavily on specific applications. However, instrumentation or probes can be easily added to the framework. The application layer (e.g., traffic engineering in CgNet or transaction processing in Client/Server) could be replaced with another specific application, so that the framework could be used to quickly generate executable prototypes and to collect useful and realistic data for various scenarios. Hence, evaluation of qualities for architectural alternatives can be enhanced and informed tradeoffs could be supported early.

Another experience is that study of patterns can be much more effective with concrete examples. CgNet shares similarities with some design patterns, even though it was not built upon well-known patterns in the first place. The main reason was that CgNet was developed by experienced software engineers and patterns were discovered from best practices anyway. Patterns document recurring solutions; however, they may not be easy to learn, especially for complicated applications like distributed computing. With a concrete, executable system, it became much easier to understand those patterns than simply studying the patterns documentation.

6. Conclusions and future work

The novelty of our approach is to advocate modeling the variation point at the architecture level with an aim to support software architecture evaluation, prototyping, and/or incremental development. Empirical study was conducted using GenVoca to build a generative framework that can be used to instantiate systems using three different architectural alternatives based on user's selection. By generating executable systems, functional requirements can be demonstrated and more realistic and concrete data can be collected to support non-functional quality assessment.

The framework can be further expanded to support other new design alternatives, if needed. One potential direction is to study architectural alternatives for multi-core systems as they have become popular. New design techniques may be developed for multi-core systems to make better use of parallelism. Hence, new patterns may be discovered in the future, which can be incorporated into the framework.

There is also a need of visual or higher-level tools to support automatic transformation to templates used in GenVoca. The modeling phase of the generative programming approach actually has captured the essence of components, their compositions, and the layered architecture as demonstrated in Fig. 5. Tools can be built to obtain the parameterization information from the user and automatically or semi-automatically transform the layered model to template representations, since the components have been captured and the programming technique has well-defined rules or grammars. With such a tool, users only need to deal with feature modeling and layered architecture design, but the template programming is hidden from the users.

Another research direction is the development and integration of front-end analysis tools. Few tools are available to software architects for conducting front-end analysis. Evaluation of some non-functional requirements, such as performance, is often difficult in practice, mainly because most software architecture evaluations are manually conducted on high-level descriptions only. There is a need to collect realistic data rapidly and more accurately using working systems. Performance data could also be stored in a performance knowledge base [41] and fed into performance models for further sensitivity or scalability analysis, even for various design variations.

Acknowledgements

We would like to thank Nortel Networks for providing us a network routing software system for research and education. We are indebted to the anonymous reviewers for their useful comments on the paper. The project is partially funded by NSERC (National Sciences and Engineering Research Council) of Canada; Grant Number RGP 251177-2002.

Appendix A. Structures for CgNet (P2P) systems using three architectural patterns

The following structure, CgSt, is created for Single Thread pattern for CgNet. Components are linked manually according to the GenVoca layered structure. Component CgNetTask, the innermost component, is linked first and then component STOtherThreadCreater is linked with component CgNetTask, and so on. At the end, component StartNodeST, starting a node process using ST, is linked with all other components.

```
struct CgSt
{
  typedef
StartNodeST<CgNetCommonSetup<CgNetOtherSetup<DestinationThread
```

```
Creater<STOtherThreadCreater<CgNetTask> > > > > cgNetSTApp;
};
```

The following structure, CgLF, is created for Leader/Followers pattern. Components are linked manually according to the GenVoca layered structure. Component CgNetTask, the innermost component, is linked first and then component LFOtherThreadCreater is linked with component CgNetTask, and so on. At the end, component StartNodeLF, starting a node process using LFs, is linked with all other components.

```
struct CgLF
{
  typedef
StartNodeLF<CgNetCommonSetup<CgNetOtherSetup<DestinationThread
Creater<LFOtherThreadCreater<CgNetTask> > > > > cgNetLFApp;
};
```

The following structure, CgLF, is created for Half-Sync/Half-Async pattern. Components are linked manually according to the GenVoca layered structure. Component CgNetTask, the innermost component, is linked first and then component HsHaOtherThreadCreater is linked with component CgNetTask, and so on. At the end, component StartNodeHsHa, starting a node process using HS/HA, is linked with all other components.

```
struct CgHsHa
{
  typedef
StartNodeHsHa<CgNetCommonSetup<CgNetHsHaSetup<HsHaDestinationThread
Creater<HsHaOtherThreadCreater<CgNetTask> > > > > cgNetHsHaApp;
};
```

Appendix B. Domain-specific language and the generator

B.1. Domain-specific language

Constant variables are defined for each architectural pattern used in the framework:

- ST – Single Thread pattern
- LF - Leader/Followers pattern
- HSHA - Half-Sync/Half-Async pattern

```
enum DesignPattern
{
     ST,
     LF,
     HSHA
};
```

Constant variables are defined for each distributed model used in the framework. Two models are defined: ClientServer and CgNet.

```
enum DistributedSystem
{
     ClientServer,
     CgNet
};
```

A structure is created for CgNet emulator. Single Thread is the default pattern.

```
template<DesignPattern pattern = ST> // ST is the default pattern
struct CgnetEmulator
{
  enum
  {
     choice = pattern,
     version = CgNet // for CgNet(P2P) application
  };
};
```

A structure is created for Client/Server. Single Thread is the default pattern.

```
template<DesignPattern pattern = ST> // ST is the default pattern
struct ClientServerArchitecure
{
  enum
  {
     choice = pattern,
     version = ClientServer // for Client/Server application
  };
};
```

B.2. Generator for Client/Server application

The generator for the Client/Server application is described as follows. User selection is passed into the generator structure and the application is built dynamically. Fig. 5 is the reference layered architecture.

```
template<class Base>
struct FRAME_WORK_CLIENTSERVER
{

  // the choice and version (see the structure defined in B.1)
  // of the application is obtained from user input or default
  enum {
    Flag = Base::choice,
    DISFlag = Base::version
  };
  // switch case for obtaining required components and
  // assembling the requested application type

    typedef SWITCH<Flag,
         CASE<ST, STInitializeBuffer<ClientServerConfig>,
         CASE<LF, InitializeBufferOthers<ClientServerConfig>,
         CASE<HSHA, InitializeBufferOthers<ClientServerConfig>
         > > > >::RET
         CSLayerOneComponent;

    typedef SharedMemoryAndMutexSetup<CSLayerOneComponent>
         CSLayerTwoComponent;

    typedef SWITCH<Flag,
         CASE<ST,CreateMainThreadST<CSLayerTwoComponent>,
         CASE<LF,DoReadLF<CSLayerTwoComponent>,
         CASE<HSHA,CreateWorkerThreadHS<CSLayerTwoComponent>
         > > > >::RET
         CSLayerThreeComponent;

    typedef SWITCH<Flag,
         CASE<ST,CreateMainThreadST<CSLayerThreeComponent>,
         CASE<LF,CreateWorkerThreadLF<CSLayerThreeComponent>,
         CASE<HSHA,CreateMainThreadHSHA<CSLayerThreeComponent>
         > > > >::RET RET;

};
```

B.3. Run-time instantiation for Client/Server model

```
// Client/Server - single thread type is built at run-time
typedef FRAME_WORK_CLIENTSERVER<ClientServerArchitecure<ST> >::RET TS;

// Client/Server - Hs/Ha is built at run-time
typedef FRAME_WORK_CLIENTSERVER<ClientServerArchitecure<HSHA> >::RET TS ;
```

```
// Client/Server - Leader/Followers type is built at run-time
typedef FRAME_WORK_CLIENTSERVER<ClientServerArchitecure<LF> >::RET TS;
```

B.4. Generator for CgNet application

The generator for the CgNet application is described as follows. User selection is passed into the generator structure and the application is built dynamically. Fig. 5 is the reference layered architecture.

```
template<class Base>
struct FRAME_WORK_CGNET
{
// One of the patterns is assigned to Flag
// One of the distributed models is assigned to DISFlag
enum {
  Flag = Base::choice,
  DISFlag = Base::version
};

// According to the pattern that is assigned to Flag above,
// any one of the three cases is selected.

   typedef SWITCH<Flag,
        CASE<ST,STOtherThreadCreater<CgNetTask>,
        CASE<LF,LFOtherThreadCreater<CgNetTask>,
        CASE<HSHA,HsHaOtherThreadCreater<CgNetTask>
        > > > >::RET
        CGLayerOneComponent;

   typedef SWITCH<Flag,
        CASE<ST,DestinationThreadCreater<CGLayerOneComponent>,
        CASE<LF,DestinationThreadCreater<CGLayerOneComponent>,
        CASE<HSHA,
             HsHaDestinationThreadCreater<CGLayerOneComponent>
        > > > >::RET
        CGLayerTwoComponent;

   typedef SWITCH<Flag,
        CASE<ST,CgNetOtherSetup<CGLayerTwoComponent>,
        CASE<LF, CgNetOtherSetup<CGLayerTwoComponent>,
        CASE<HSHA, CgNetHsHaSetup<CGLayerTwoComponent>
        > > > >::RET
        CGLayerThreeComponent;

   typedef CgNetCommonSetup<CGLayerThreeComponent>
        CGLayerFourComponent;

   typedef SWITCH<Flag,
        CASE<ST, StartNodeST<CGLayerFourComponent>,
        CASE<LF, StartNodeLF<CGLayerFourComponent>,
        CASE<HSHA, StartNodeHsHa<CGLayerFourComponent>
        > > > >::RET RET;
};
```

B.5. Run-time instantiation for CgNet application

```
// CgNet - single thread pattern is built at run-time
typedef FRAME_WORK_CGNET<CgnetEmulator<ST> >::RET     CG_ST;

//CgNet - HS/HA pattern is built at run-time
typedef FRAME_WORK_CGNET<CgnetEmulator<HSHA> >::RET   CG_HSHA;
```

```
// CgNet - Leader/Followers pattern is built at run-time
typedef FRAME_WORK_CGNET<CgnetEmulator<LF> >::RET     CG_LF
```

References

[1] A. Alhussaini, B. Balasubramaniam, P. Chandrabose, A. Kasinathan, Software restructuring and performance evaluation, in: Project Report, Department of Systems & Computer Engineering, Carleton University; 2004.
[2] J.E. Bardram, H.B. Christensen, K.M. Hansen, Architectural prototyping: An approach for grounding architectural design and learning, in: Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture; 2004; pp. 15–24.
[3] D. Batory, B.J. Geraci, Composition validation and subjectivity in Genvoca generators, IEEE Trans. on Software Engineering 23 (2) (1997) 67–82.
[4] D. Batory, S. O'Malley, The design and implementation of hierarchical software systems with reusable components, ACM Transactions on Software Engineering and Methodology 1 (4) (2002) 355–398.
[5] D. Batory, J.D. Sarvela, A. Rauschmayer, Scaling step-wise refinement, IEEE Transactions on Software Engineering 30 (6) (2004) 355–371.
[6] D. Batory, Multi-level models in model driven development, product-lines, and metaprogramming, IBM Systems Journal 45 (3) (2006) 1–13.
[7] D. Batory, Program refactoring, program synthesis, and model-driven development, Invited Presentation at the European Joint Conf. on Theory and Practice of Software Compiler Construction Conf; 2007.
[8] Bayer J., et al. Process family engineering in automotive control systems – a case study, in: Proceedings of the 1st Generative Programming and Component Engineering for QoS Provisioning in Distributed Systems (GPCE4QoS) Workshop; Portland, OR, 2006.
[9] F. Bachmann, L. Bass, M. Klein, Preliminary design of arche: A software architecture design assistant, Technical Report CMU/SEI-2003-TR-021, Software Engineering Institute; Sept. 2003.
[10] J. Bezivin, Model driven engineering: principles, scope, deployment, and applicability, GTTSE 2005.
[11] G. Booch, A. Brown, S. Iyengar, J. Rumbaugh, B. Selic, The IBM MDA manifesto, The MDA Journal (2004).
[12] J. Bosch, Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach, Addison-Wesley, Reading, MA, 2000.
[13] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture: A System of Patterns, Wiley, 1996.
[14] D.-J. Chen, M.-J. Tsai, S.-T. Yang, UI design pattern generator for pervasive devices, in: Proceedings of the International Conference on Software Engineering and Knowledge Engineering; Taipei, Taiwan, 2005; pp. 360–365.
[15] K. Czarnecki, U.W. Eisenecker, Generative Programming Methods, Tools, and Applications, Addison Wesley, 2000.
[16] K. Czarnecki, A. Wasowski, Feature diagrams and logics: There and back again, in: Proceedings of the 11th International Software Product Line Conference, 2007; pp. 23–34.
[17] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.
[18] G.T. Heineman, W.T. Councill, Component Based Software Engineering: Putting the Pieces Together, Addison-Wesley, 2001.
[19] J. Hillston, A Compositional Approach to Performance Modelling, Cambridge University Press, 1996.
[20] C. Hobbs, G. Young, CgNet: A User's Guide & Designer's Manual, Nortel Networks, 2001.
[21] K. Kang, S. Cohen, J. Hess, W. Nowak, S. Peterson, Feature-oriented domain analysis (FODA) feasibility study, Technical Report CMU/SEI-90-TR-21, Software Engineering, Pittsburgh, PA: Institute, Carnegie Mellon University, Nov. 1990.
[22] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, The architecture tradeoff analysis method, in: Proceedings of the 4th International Conference on Engineering of Complex Computer Systems, 1998, pp. 68–78.
[23] J.-C. Lee, X. Zhang, Performance investigation of a network system on different linux kernels, Project Report, Dept. of Systems & Computer Eng., Carleton Univ., Ottawa, Canada, 2004.
[24] J. Loeckx, H.D. Ehrich, D. Wolf, Specification of Abstract Data Types, John Wiley & Sons Ltd, Chichester, UK, 1996.
[25] R. Lopez-Herrejon, D. Batory, C. Lengauer, A disciplined approach to aspect composition, in: Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation; 2006; pp. 66–77.
[26] C.-H. Lung, K. Kalaichelvan, A quantitative approach to software architecture sensitivity analysis, International Journal of Software Engineering and Knowledge Engineering 10 (1) (2000) 97–114.
[27] C.-H. Lung, Variability Analysis for Communications Software, in: Proceedings of the International Workshop on Software Variability Management (SVM), International Conference on Software Eng. (2003) 30–33.
[28] C.-H. Lung, Q. Zhao, H. Xu, H. Mar, P. Kanagaratnam, Experience of communications software evolution and performance improvement with patterns, in: Proceedings of IASTED Software Engineering, 2004, pp. 321–326.
[29] C.-H. Lung, Q. Zhao, Pattern-oriented reengineering of a network system, Journal of Systemics, Cybernetics and Informatics 2 (5) (2004).
[30] C.-H. Lung, et al. Architecture-centric software generation: An experimental study on distributed systems, in: Proc. of Generative Programming and Component Engineering for QoS Provisioning in Distributed Systems, 2006.
[31] F. Martensson, H. Grahn, M. Mattsson, Prototype-based software architecture evaluation – component quality attribute evaluation, in: Proceedings of the 4th Conference on Software Engineering Research and Practice in Sweden; 2004; pp. 11–17.
[32] L.M. Northrop, P.C. Clements, A framework for software product line practice, Version 5.0. Software Engineering Institute, Carnegie Mellon University; 2005.
[33] P. Rajeswaran, S. Sivadas, T. Sivabalasingam, Development of a C++ generative framework for distributed systems. Project Report, Department of Systems & Computer Engineering, Carleton University, Ottawa, Canada, 2004.
[34] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann, Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Wiley, 2000.
[35] C.U. Smith, L.G. Williams, Performance Solutions A Practical Guide to Creating Responsive and Scalable Software, Addison-Wesley, 2001.
[36] V.P. Singhal, A programming language for writing domain-specific software system generators, Ph.D. Thesis, Department of Computer Science, Univ. of Texas at Austin, Sept. 1996.
[37] J. Sztipanovits, G. Karsai, Generative programming for embedded systems, in: Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering; 2002; pp. 32–49.
[38] S. Trujillo, D. Batory, O. Diaz, Feature oriented model driven development: A case study for portlets, in: Proc. of the 29th International Conference on Software Engineering, 2007; pp. 44-53.
[39] S. Trujillo, M. Azanza, O. Diaz, Generative metaprogramming, in: Proc of the 6th Int. Conf. on Generative Programming and Component Engineering, Oct. 2007; pp. 105–114.
[40] C.M. Woodside, J.E Neilson, D.C. Petriu, S. Majumdar, The stochastic rendezvous network model for performance of synchronous client-server-like distributed software, IEEE Transactions on Computers 44 (1) (1995) 20–34.
[41] C.M. Woodside, G. Franks, D.C. Petriu, The future of software performance engineering, in: Proc. of the 29th International Conference on Software Engineering; 2007; pp. 171–187.
[42] P. Wu, C.M. Woodside, C.-H. Lung, Compositional performance modeling for peer-to-peer routing software, in: Proc. of the IEEE Int. Performance Computing and Communications Conference; 2004; pp. 231–238.